Implementierung von x^y-Funktion in x86_64 Assembler

von Maksym Bondarenko, Ihor Kudryk,
Aiina Tikhonova

Übersicht

- Darstellung und Problemstellung
- Lösungsmöglichkeiten
- Alternativen
- Implementierung
- Beispiel
- Umgesetzte Optimierungen
- Genauigkeit
- Performanz
- Tests
- Zusammenfassung

Darstellung und Problemstellung

<u>Aufgabe:</u> Implementierung von Potenzfunktion $f(x, y) = x^y$ in 86_64 Assembler. Dabei ist es für **rationale** y x > 0 anzunehmen. Für **ganze** y gilt keine Einschränkung für x.

<u>Frage:</u> wie soll **f** implementiert werden, wenn Eingaben auch als Gleitkommazahlen angenommen werden können? Insbesondere ist es für y-Werte wichtig.

Lösungsansatz

1. Darstellung mit Hilfe von Exponentialfunktion:

$$x^y = e^{y \cdot ln(x)}$$

2. Approximation für ***ln(x)*** durch die Modifizierte **Taylor-Entwicklung** für $e^{y \cdot ln(x)}$ - **Logarithmuserweiterung**
3. Berechnung des Ergebnisses mit Hilfe von **Tayloransatzes**

Logarithmuserweiterung

für **n = 10** äquidistante Stützpunkte u_i , i $\in$ {0..9}:

$$ln(x) = \int_1^x y(u)du \approx \frac{h}{3}[y(u_0) + y(u_{10}) + 2(y(u_2) + y(u_4) + y(u_6) + y(u_8))$$
$$+4(y(u_1) + y(u_3) + y(u_5) + y(u_7) + y(u_9))]$$

wobei $h = \frac{b-a}{2n}$

und $y(u) = \frac{1}{u}$

Tayloransatz

Berechnen e^z, wo $z = y\ ln(x)$:

$$e^z = 1 + \frac{z}{1!} + \frac{z^2}{2!} + \frac{z^3}{3!} + \ldots$$
$$= 1 + \frac{z}{1} \cdot (1 + \frac{z}{2} \cdot (1 + \frac{z}{3} \cdot (\ldots)))$$

Lösungsalternativen

Anstatt der Logarithmuserweiterung könnte eine **Approximation** durch die **Simpsonregel** angewendet werden, um den gesuchten Logarithmuswert anzunähern.

Dieser Ansatz erlaubt die Ableitungsfunktion von ln(x) näherungsweise zu integrieren.

$$\ln a = \int_1^a \frac{1}{x}\, dx$$

Implementierung

Es stehen weitere Funktionen zur Verfügung:

- "*f*" ist unsere Potenzfunktion und liefert das Endergebnis
- "***pow_int***" berechnet x^g, wobei **g** abgerundeter **y**-Wert ist
- "***ln***" berechnet den Wert von **ln(x)** mithilfe der Logarithmuserweiterung
- "***exponent***" berechnet das Ergebnis des Taylöransatzes

$$x^y = e^{y \cdot ln(x)}$$

Umgesetzte Optimierungen

$$x^y = x^{g,r} = x^g \cdot x^{0,r}$$

Die Potenz wird aufgespaltet auf **ganz-** und **gleitkomma-**zahlige Teilen.

Somit gilt: ***0 ≤ r < 1***

Wenn die nächste Iteration der Schleife den Wert des Ergebnisses nicht ändert - wird die **Schleife unterbrochen** und der Wert wird in einer **schnelleren Zeit** erhalten

Beispiel

$x = 1.3; y = 4.2$

1. y wird aufgespaltet: $1.3^{4.2} = 1.3^{4} * 1.3^{0.2}$
2. Mit ganzer Potenz wird es schneller berechnet: $1.3^{4} = 2.8561$
3. Tayloransatz: $1.3^{0.2} = e^{0.2\ln(1.3)}$
4. Logarithmuserweiterung: $\ln(1.3) = 0.262364264$
5. Potenz: $0.2 * \ln(1.3) = 0.052472852$
6. $e^{0.2\ln(1.3)} = e^{0.052472852} = 1.053873952$
7. 1.053873952 * 2.8561 = 3.009969394 - Endergebnis

Verbesserungen und Alternativen

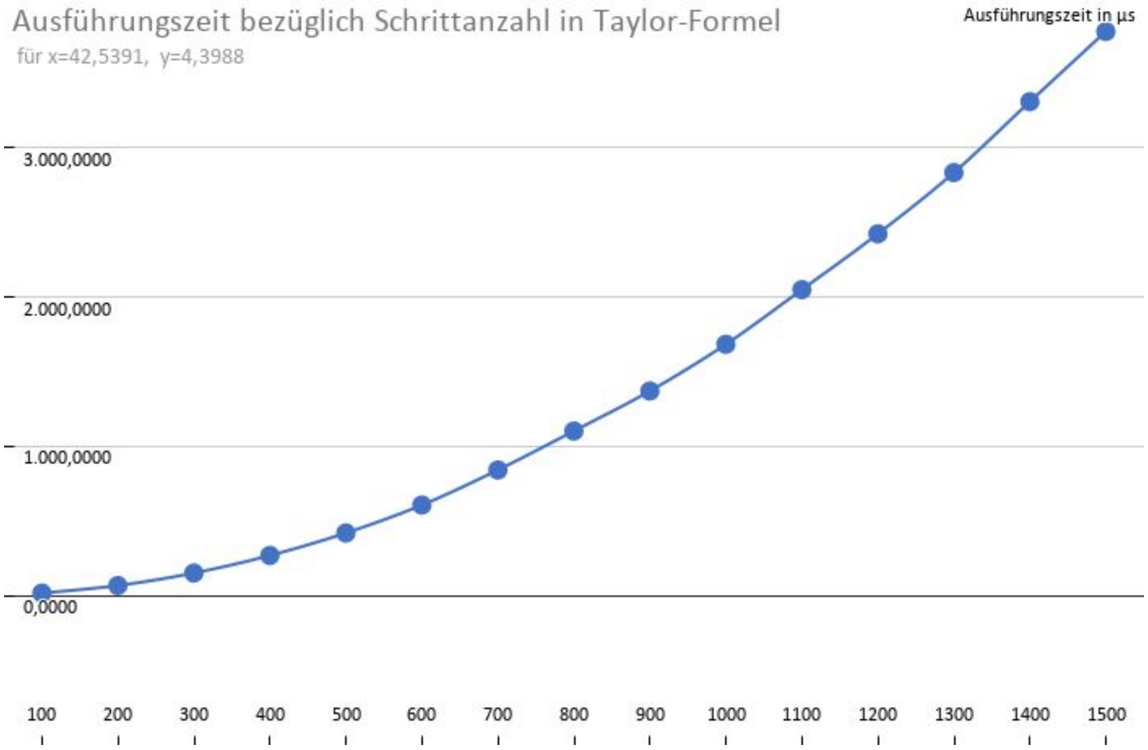
- Binäre Exponentiation
- Auswahl des Algorithmus (Simpson/Taylor) in Abhängigkeit von den Eingabewerten

Genauigkeit

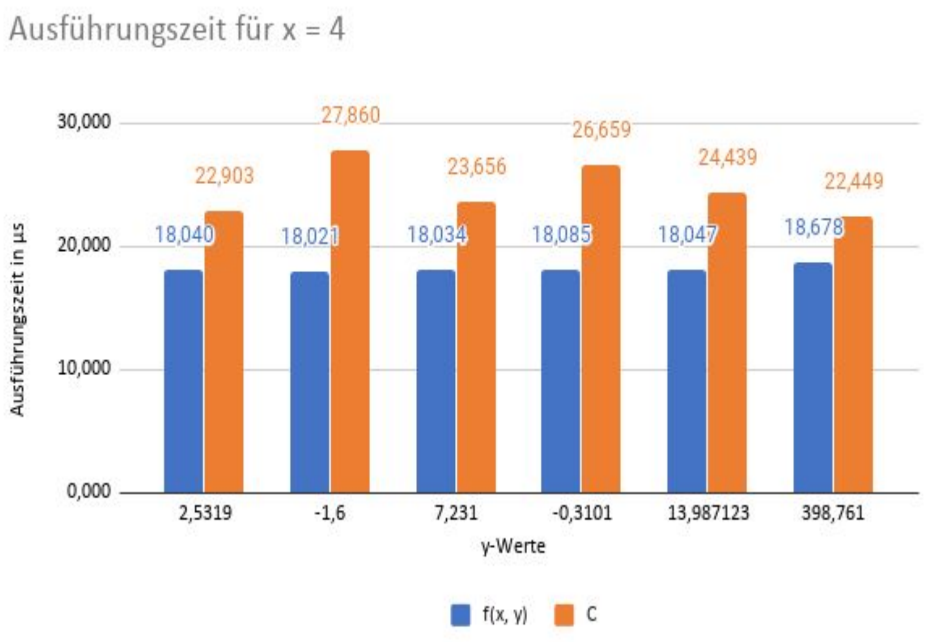
Genauigkeitsvergleich mit Schritt = 100									
X\Y (%)	2,4253	3	-5	420	0,67	-0,81	347,63	-382,74	73,31339
0	0	0	0	0	0	0	0	0	0
-1		0	0	0					
1	0	0	0	0	0	0	0	0	0
3,4	0	0	0	1,39E-13	0	0	1,71E-13	0	4,92E-14
-364,5		0	0	0					
1,24	1,32E-14	0	0	5,78E-13	0	0	3,37E-13	0	3,96E-14
-149		0	0	0					
-4		0	0	0					
1200	29,71%	0	0	0	-42,61%	95,70%	0	0	22,88%
4242	54,98%	0	0	0	-71,55%	357,10%	0	0	44,46%

Genauigkeitsvergleich mit Schritt = 10.000									
X\Y (%)	2,4253	3	-5	420	0,67	-0,81	347,63	-382,74	73,31339
0	0	0	0	0	0	0	0	0	0
-1		0	0	0					
1	0	0	0	0	0	0	0	0	0
3,4	0	0	0	0	0	0	1,71E-13	0	4,92E-14
-364,5		0	0	0					
1,24	1,32E-14	0	0	5,78E-13	0	0	3,37E-13	0	3,96E-14
-149		0	0	0					
-4		0	0	0					
1200	1,12E-12	0	0	0	0	0	0	0	8,39E-13
4242	3,30E-04	0	0	0	-0,50%	-0,60%	0	0	0,02%

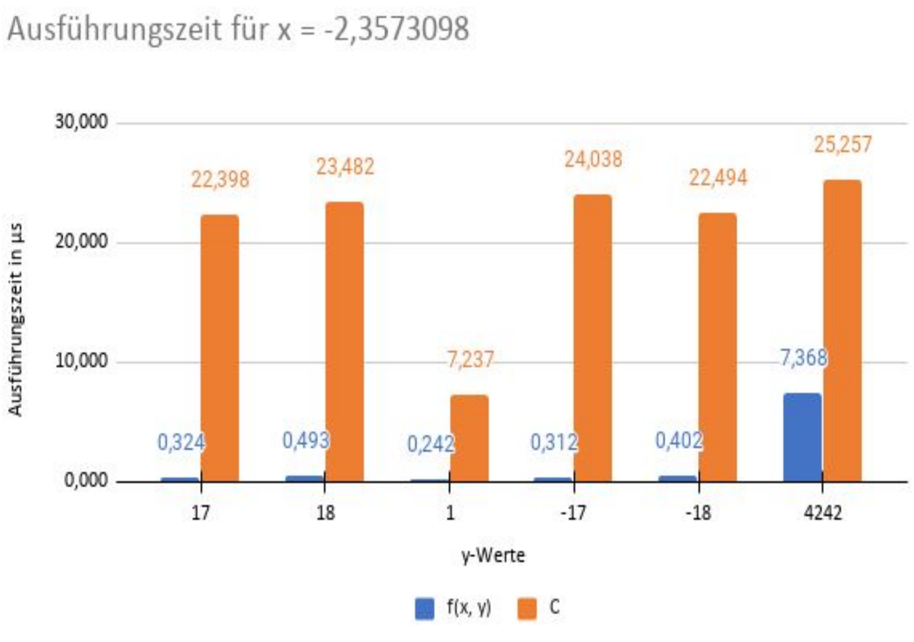
Ausführungszeit



Ausführungszeit

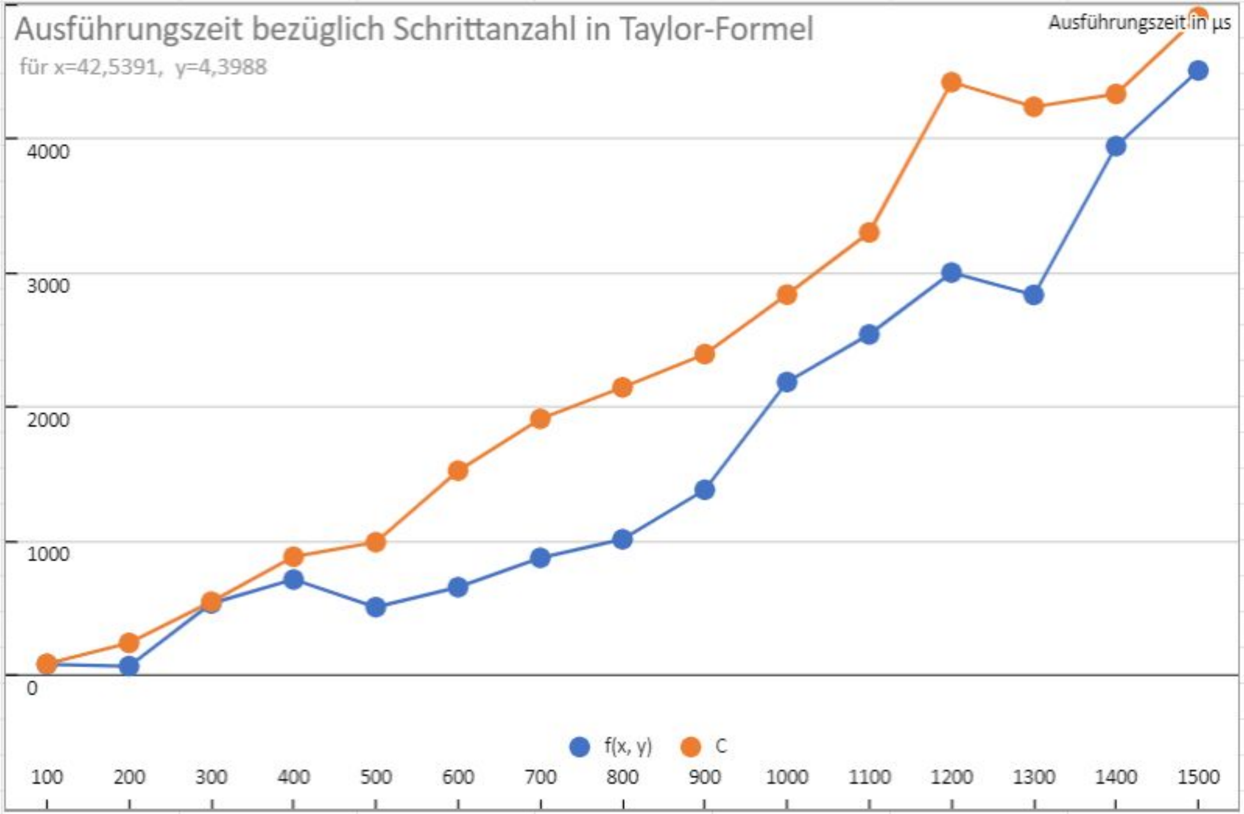


mit rationalem y-Wert



mit ganzzahligem y-Wert

Ausführungszeit



Automatische Tests

Für die Performanzanalyse wurden 2 automatische Tests verwendet

Dabei wird eine Textdatei erwartet, wovon die Werte für Performanzanalyse abgelesen werden. Dieser Test erfolgt automatisch und erzeugt 4 neue Dateien.

Auf Performanz wird jeweils Alternative Implementierung in Assembler, Implementierung in Assembler, Implementierung in C , pow(double x, double y) aus der Standardbibliothek

Diese Tests ermöglichen eine schnelle und sichere Analyse über mit mehreren Durchschnittswerte

Zusammenfassung

- Im Vergleich zu nicht optimiertem C-Code und der Bibliotheksfunktion (-O1) liefert unsere Implementierung im Durchschnitt genaue Werte in kürzerer Zeit
- Verglichen mit dem optimierten C-Code (-O3) liefert unsere Implementierung genauere Ergebnisse und weniger zeitaufwändige Arbeit.
- Im Vergleich zur optimierten Bibliotheksfunktion liefert unsere Implementierung fast die gleichen exakten Werte, jedoch über einen längeren Zeitraum. Dies kann jedoch durch die Umsetzung der zuvor beschriebenen Verbesserungen korrigiert werden.
- Die Potenzrechnung mithilfe der Taylorreihe-Entwicklung ist wünschenswert, wenn eine schnelle Implementierung der Potenzfunktion für keine großen Double-Werte erforderlich ist.

120+ hours of coding, 1000+ lines of code, 18,9 liters of coffee, 8 pizzas and only 3 enthusiasts...